GAME PLAYING

CMSC 421, Chapter 6
Finite perfect-information zero-sum games

**Finite**: finitely many agents, actions, states

**Perfect information**: every agent knows the current state, all of the actions, and what they do
No simultaneous actions – players move one-at-a-time

**Constant-sum**: regardless of how the game ends, \( \sum \{ \text{agents' utilities} \} = k \).
For every such game, there’s an equivalent game in which \( k = 0 \).
Thus constant-sum games usually are called **zero-sum** games

**Examples:**

- **Deterministic**: chess, checkers, go, othello (reversi), connect-four, qubic, mancala (awari, kalah), 9 men’s morris (merelles, morels, mill)
- **Stochastic**: backgammon, monopoly, yahtzee, parcheesi, roulette, craps

We’ll start with deterministic games
Outline

◇ A brief history of work on this topic
◇ The minimax theorem
◇ Game trees
◇ The minimax algorithm
◇ $\alpha$-$\beta$ pruning
◇ Resource limits and approximate evaluation
A brief history

1846 (Babbage): machine to play tic-tac-toe

1928 (von Neumann): minimax theorem

1944 (von Neumann & Morgenstern): backward-induction algorithm (produces perfect play)

1950 (Shannon): minimax algorithm (finite horizon, approximate evaluation)

1951 (Turing): program (on paper) for playing chess

1952–7 (Samuel): checkers program, capable of beating its creator

1956 (McCarthy): pruning to allow deeper search

1957 (Bernstein): first complete chess program, on an IBM 704 vacuum-tube computer, could examine about 350 positions/minute
**A brief history, continued**

1967 (Greenblatt): first program to compete in human chess tournaments:

- 3 wins, 3 draws, 12 losses

1992 (Schaeffer): Chinook won the 1992 US Open checkers tournament

1994 (Schaeffer): Chinook became world checkers champion;

- Tinsley (human champion) withdrew for health reasons

1997 (Hsu et al): Deep Blue won 6-game chess match against world chess champion Gary Kasparov

2007 (Schaeffer et al, 2007): Checkers solved: with perfect play, it’s a draw. This took $10^{14}$ calculations over 18 years
Basics

◊ A strategy specifies what an agent will do in every possible situation
◊ Strategies may be pure (deterministic) or mixed (probabilistic)

Suppose agents $A$ and $B$ use strategies $s$ and $t$ to play a two-person zero-sum game $G$. Then

- $A$’s expected utility is $U_A(s, t)$
  From now on, we’ll just call this $U(s, t)$

- Since $G$ is zero-sum, $U_B(s, t) = -U(s, t)$

Instead of $A$ and $B$, we’ll call the agents Max and Min

Max wants to maximize $U$ and Min wants to minimize it
The Minimax Theorem (von Neumann, 1928)

**Minimax theorem:** Let $G$ be a two-person finite zero-sum game with players Max and Min. Then there are strategies $s^*$ and $t^*$, and a number $V_G$ called $G$'s *minimax value*, such that

- If Min uses $t^*$, Max’s expected utility is $\leq V_G$, i.e., $\max_s U(s, t^*) = V_G$
- If Max uses $s^*$, Max’s expected utility is $\geq V_G$, i.e., $\min_t U(s^*, t) = V_G$

**Corollary 1:** $U(s^*, t^*) = V_G$.

**Corollary 2:** If $G$ is a perfect-information game, then there are *pure* strategies $s^*$ and $t^*$ that satisfy the theorem.
Game trees

The name *game tree* comes from AI. Mathematical game theorists call this the *extensive form* of a game.

Root node ⇔ the initial state

Children of a node ⇔ the states a player can move to

The name *game tree* comes from AI. Mathematical game theorists call this the *extensive form* of a game.

Root node ⇔ the initial state

Children of a node ⇔ the states a player can move to

The name *game tree* comes from AI. Mathematical game theorists call this the *extensive form* of a game.

Root node ⇔ the initial state

Children of a node ⇔ the states a player can move to
To construct a pure strategy for Max:

- At each node where it’s Max’s move, choose one branch
- At each node where it’s Min’s move, include all branches

Let \( b \) = the *branching factor* (max. number of children of any node)

\[ h = \text{the tree’s height} \] (max. depth of any node)

The number of pure strategies for Max \( \leq b^{[h/2]} \),
with equality if every node of height \( < h \) node has \( b \) children
Strategies on game trees

To construct a pure strategy for Min:

- At each node where it’s Min’s move, choose one branch
- At each node where it’s Max’s move, include all branches

The number of pure strategies for Min $\leq b^{\lceil h/2 \rceil}$
with equality if every node of height $< h$ node has $b$ children
Finding the best strategy

Brute-force way to find Max’s and Min’s best strategies:

Construct the sets $S$ and $T$ of all of Max’s and Min’s pure strategies, then choose

\[
\begin{align*}
    s^* &= \arg \max_{s \in S} \min_{t \in T} U_{\text{Max}}(s, t) \\
    t^* &= \arg \min_{t \in T} \max_{s \in S} U_{\text{Max}}(s, t)
\end{align*}
\]

Complexity analysis:

- Need to construct and store $O(b^{h/2} + b^{h/2}) = O(b^{h/2})$ strategies
- Each strategy is a tree that has $O(b^{h/2})$ nodes
- Thus space complexity is $O(b^{h/2}b^{h/2}) = O(b^h)$
- Time complexity is slightly worse

But there’s an easier way to find the strategies
Minimax Algorithm

Compute a game’s minimax value recursively from the minimax values of its subgames:

```
function Minimax(s) returns a utility value
    if s is a terminal state then return Max’s payoff at s
    else if it is Max’s move in s then
        return \max\{Minimax(result(a, s)) : a is applicable to s\}
    else return \min\{Minimax(result(a, s)) : a is applicable to s\}
```

To get the next action, return argmax and argmin instead of max and min
Properties of the minimax algorithm

*Is it sound?* I.e., when it returns answers, are they correct?
Properties of the minimax algorithm

*Is it sound?* I.e., when it returns answers, are they correct?
Yes (can prove this by induction)

*Is it complete?* I.e., does it always return an answer when one exists?
Properties of the minimax algorithm

Is it sound? I.e., when it returns answers, are they correct?
Yes (can prove this by induction)

Is it complete? I.e., does it always return an answer when one exists?
Yes on finite trees (e.g., chess has specific rules for this).

Space complexity?
Properties of the minimax algorithm

Is it sound?  I.e., when it returns answers, are they correct?
   Yes (can prove this by induction)

Is it complete?  I.e., does it always return an answer when one exists?
   Yes on finite trees (e.g., chess has specific rules for this).

Space complexity?  $O(bh)$, where $b$ and $h$ are as defined earlier

Time complexity?
Properties of the minimax algorithm

*Is it sound?*  I.e., when it returns answers, are they correct?
   Yes (can prove this by induction)

*Is it complete?*  I.e., does it always return an answer when one exists?
   Yes on *finite* trees (e.g., chess has specific rules for this).

*Space complexity?*  $O(bh)$, where $b$ and $h$ are as defined earlier

*Time complexity?*  $O(b^h)$

For chess, $b \approx 35$, $h \approx 100$ for “reasonable” games

$35^{100} \approx 10^{135}$ nodes

This is about $10^{55}$ times the number of particles in the universe (about $10^{87}$)

$\Rightarrow$ no way to examine every node!

But do we really need to examine every node?
Pruning example 1

MAX

MIN

3

3 12 8

3

2

2

XX

3

CMSC 421, Chapter 6  18
Max will never move to this node, because Max can do better by moving to the first one.

Thus we don’t need to figure out this node’s minimax value.
Pruning example 1

This node might be better than the first one
Pruning example 1

It still might be better than the first one
Pruning example 1

No, it isn’t
Pruning example 2

Same idea works farther down in the tree

Max won’t move to $e$, because Max can do better by going to $b$
Don’t need $e$’s exact value, because it won’t change $\text{minimax}(a)$
So stop searching below $e$
**Alpha-beta pruning**

Start a minimax search at node \( c \)

Let \( \alpha = \) biggest lower bound on any ancestor of \( f \)

\[
\alpha = \max(-2, 4, 0) = 4 \quad \text{in the example}
\]

If the game reaches \( f \), Max will get utility \( \leq 3 \)

To reach \( f \), the game must go through \( d \)

But if the game reaches \( d \), Max can get utility \( \geq 4 \) by moving off of the path to \( f \)

So the game will never reach \( f \)

We can stop trying to compute \( u^*(f) \), because it can’t affect \( u^*(c) \)

This is called an **alpha cutoff**
Alpha-beta pruning

Start a minimax search at node $a$

Let $\beta =$ smallest upper bound on any ancestor of $d$

$$\beta = \min(5, -2, 3) = -2$$ in the example

If the game reaches $d$, Max will get utility $\geq 0$

To reach $d$, the game must go through $b$

But if the game reaches $b$, Min can make Max’s utility $\leq -2$ by moving off the path to $d$

So the game will never reach $d$

We can stop trying to compute $u^*(d)$, because it can’t affect $u^*(a)$

This is called a beta cutoff
**The alpha-beta algorithm**

```plaintext
function ALPHA-BETA(s, α, β) returns a utility value
    inputs: s, current state in game
            α, the value of the best alternative for MAX along the path to s
            β, the value of the best alternative for MIN along the path to s
    if s is a terminal state then return Max’s payoff at s
    else if it is Max’s move at s then
        v ← −∞
        for every action a applicable to s do
            v ← max(v, ALPHA-BETA(result(a, s), α, β))
            if v ≥ β then return v
            α ← max(α, v)
    else
        v ← ∞
        for every action a applicable to s do
            v ← min(v, ALPHA-BETA(result(a, s), α, β))
            if v ≤ α then return v
            β ← min(β, v)
    return v
```
\( \alpha \)-\( \beta \) pruning example

\[ \begin{align*}
\alpha &= -\infty \\
\beta &= \infty
\end{align*} \]
$\alpha$-$\beta$ pruning example

$\alpha = -\infty, 7$
$\beta = \infty$

$\alpha = -\infty$
$\beta = \infty$

CMSC 421, Chapter 6  28
\( \alpha - \beta \) pruning example
**$\alpha$-$\beta$ pruning example**

\[
\begin{align*}
\alpha &= -\infty \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= \infty \\
\beta &= 7
\end{align*}
\]

\[
\begin{align*}
\alpha &= -\infty \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= \infty \\
\beta &= 7
\end{align*}
\]
**α-β pruning example**

```
α = −∞  7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
X   7
α = −∞
β = ∞
```

```
5
```

```
5
```

```
5 -3
```

```
e 
 f 
 g 
 h 
 alpha cutoff
```

```
7
m
 j 
 i 
 k 
 l 
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```

```
α = 7
β = ∞
```
\( \alpha - \beta \) pruning example

\[ a = -\infty \quad 7 \]
\[ \beta = \infty \]

\[ a = 7 \quad \beta = \infty \]

\[ a = 7 \quad \beta = \infty \]

\[ a = 7 \quad \beta = \infty \]

\[ a = 7 \quad \beta = \infty \]
$\alpha$-$\beta$ pruning example
\(\alpha - \beta\) pruning example
\( \alpha - \beta \) pruning example

\[ \begin{align*}
\alpha &= -\infty \quad 7 \\
\beta &= \infty
\end{align*} \]

\[ \begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*} \]
$\alpha - \beta$ pruning example

\begin{itemize}
  \item $\alpha = -\infty$ \hspace{1cm} $\beta = \infty$
  \item $\alpha = 7$ \hspace{1cm} $\beta = \infty$
  \item $\alpha = 7$ \hspace{1cm} $\beta = 8$
  \item $\alpha = 7$ \hspace{1cm} $\beta = 8$
\end{itemize}
Properties of $\alpha - \beta$

$\alpha - \beta$ is a simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

- if $\alpha \leq \text{minimax}(s) \leq \beta$, then alpha-beta returns $\text{minimax}(s)$
- if $\text{minimax}(s) \leq \alpha$, then alpha-beta returns a value $\leq \alpha$
- if $\text{minimax}(s) \geq \beta$, then alpha-beta returns a value $\geq \beta$

If we start with $\alpha = -\infty$ and $\beta = \infty$, then alpha-beta will always return $\text{minimax}(s)$

Good move ordering can enable us to prune more nodes.

Best case is if
- at nodes where it’s Max’s move, children are largest-value first
- at nodes where it’s Min’s move, children are smallest-value first

In this case time complexity $= O(b^{h/2})$ $\Rightarrow$ doubles the solvable depth

Worst case is the reverse
In this case, $\alpha - \beta$ will search every node
Resource limits

Even with alpha-beta, it can still be infeasible to search the entire game tree (e.g., recall chess has about $10^{135}$ nodes)

⇒ need to limit the depth of the search

Basic approach: let $d$ be a positive integer
Whenever we reach a node of depth $> d$
  • If we’re at a terminal state, then return Max’s payoff
  • Otherwise return an estimate of the node’s utility value, computed by a static evaluation function
$\alpha - \beta$ with a bound $d$ on the search depth

function $\text{Alpha-Beta}(s, \alpha, \beta, d)$ returns a utility value

inputs: $s, \alpha, \beta$, same as before

$d$, an upper bound on the search depth

if $s$ is a terminal state then return Max’s payoff at $s$
else if $d = 0$ then return $\text{Eval}(s)$
else if it is Max’s move at $s$ then
  $v \leftarrow -\infty$
  for every action $a$ applicable to $s$ do
    $v \leftarrow \max(v, \text{Alpha-Beta}(\text{result}(a, s), \alpha, \beta, d - 1))$
  if $v \geq \beta$ then return $v$
  $\alpha \leftarrow \max(\alpha, v)$
else
  $v \leftarrow \infty$
  for every action $a$ applicable to $s$ do
    $v \leftarrow \min(v, \text{Alpha-Beta}(\text{result}(a, s), \alpha, \beta, d - 1))$
  if $v \leq \alpha$ then return $v$
  $\beta \leftarrow \min(\alpha, v)$
return $v$
Evaluation functions

\textbf{Eval}(s) \text{ is supposed to return an approximation of } s \text{’s minimax value}

\textbf{Eval} \text{ is often a weighted sum of } \textbf{features}

\textbf{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)

\begin{align*}
\text{Black to move} & \quad \text{White to move} \\
\text{White slightly better} & \quad \text{Black winning}
\end{align*}

E.g.,

\begin{align*}
1 & (\text{number of white pawns} - \text{number of black pawns}) \\
+ 3 & (\text{number of white knights} - \text{number of black knights}) \\
+ \ldots
\end{align*}
Exact values for $\text{Eval}$ don’t matter

Behavior is preserved under any \textbf{monotonic} transformation of $\text{Eval}$

Only the order matters:

In deterministic games, payoff acts as an \textit{ordinal utility} function
Discussion

Deeper lookahead (i.e., larger depth bound $d$) usually gives better decisions.

Exceptions do exist:
- Main result in my PhD dissertation (30 years ago!):
  “pathological” games in which deeper lookahead gives worse decisions.
- But such games hardly ever occur in practice.

Suppose we have 100 seconds, explore $10^4$ nodes/second
  $\Rightarrow 10^6 \approx 35^{8/2}$ nodes per move
  $\Rightarrow \alpha-\beta$ reaches depth 8 $\Rightarrow$ pretty good chess program

Some modifications that can improve the accuracy or computation time:
  * node ordering (see next slide)
  * quiescence search
  * biasing
  * transposition tables
  * thinking on the opponent’s time
  ...

CMSC 421, Chapter 6   42
Node ordering

Recall that I said:

Best case is if
♦ at nodes where it’s Max’s move, children are largest first
♦ at nodes where it’s Min’s move, children are smallest first
In this case time complexity = $O(b^{h/2}) \Rightarrow$ doubles the solvable depth

Worst case is the reverse

How to get closer to the best case:
♦ Every time you expand a state $s$, apply $\text{Eval}$ to its children
♦ When it’s Max’s move, sort the children in order of largest $\text{Eval}$ first
♦ When it’s Min’s move, sort the children in order of smallest $\text{Eval}$ first
Quiescence search and biasing

In a game like checkers or chess
The evaluation is based greatly on material pieces
It’s likely to be inaccurate if there are pending captures
e.g., if someone is about to take your queen

◊ Search deeper to reach a position where there aren’t pending captures
   Evaluations will be more accurate here

But it creates another problem

◊ You’re searching some paths to an even depth, others to an odd depth

◊ Paths that end just after your opponent’s move
   will generally look worse than paths that end just after your move

◊ Add or subtract a number called the “biasing factor” to try to fix this
Transposition tables

Often there are multiple paths to the same state (i.e., the state space is a really graph rather than a tree)

Idea:
♦ when you compute $s$’s minimax value, store it in a hash table
♦ visit $s$ again $\Rightarrow$ retrieve its value rather than computing it again

The hash table is called a transposition table

Problem: far too many states to store all of them $s$

Store some of the states, rather than all of them

Try to store the ones that you’re most likely to need
Thinking on the opponent’s time

Current state $s$, children $s_1, \ldots, s_n$

Compute their minimax values, move to the one that looks best
say, $s_i$

You computed $s_i$’s minimax value as the minimum of the values of its children, $s_{i1}, \ldots, s_{im}$

Let $s_{ij}$ be the one that has the smallest minimax value
That’s where the opponent is most likely to move to

Do a minimax search below $s_{ij}$ while waiting for the opponent to move

If your opponent moves to $s_{ij}$ then you’ve already done a lot of the work of figuring out your next move
Game-tree search in practice

**Checkers**: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.

Checkers was *solved* in April 2007: from the standard starting position, both players can guarantee a draw with perfect play. This took $10^{14}$ calculations over 18 years. Checkers has a search space of size $5 \times 10^{20}$.

**Chess**: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

**Othello**: human champions refuse to compete against computers, who are too good.

**Go**: until recently, human champions didn’t compete against computers because the computers were too *bad*. But that has changed . . .
Game-tree search in the game of go

A game tree’s size grows exponentially with both its depth and its branching factor.

Go is much too big for a normal game-tree search:
- branching factor = about 200
- game length = about 250 to 300 moves
- number of paths in the game tree = $10^{525}$ to $10^{620}$

For comparison:
- Number of atoms in universe = about $10^{80}$
- Number of particles in universe = about $10^{87}$
Game-tree search in the game of go

During the past couple years, go programs have gotten much better

Main reason: Monte Carlo roll-outs

Basic idea: do a minimax search of a randomly selected subtree

At each node that the algorithm visits,

♦ It randomly selects some of the children
  There are some heuristics for deciding how many

♦ Calls itself recursively on these, ignores the others
Forward pruning in chess

Back in the 1970s, some similar ideas were tried in chess.

The approach was called **forward pruning**.
Main difference: select the children heuristically rather than randomly.
It didn’t work as well as brute-force alpha-beta, so people abandoned it.

Why does a similar idea work so much better in go?
Perfect-information nondeterministic games

Backgammon: chance is introduced by dice
function *EXPECTIMINIMAX*($s$) returns an expected utility

if $s$ is a terminal state then return Max’s payoff at $s$

if $s$ is a “chance” node then

\[
\text{return } \sum_{s'} P(s'|s) \text{EXPECTIMINIMAX}(s')
\]

else if it is Max’s move at $s$ then

\[
\text{return } \max \{\text{EXPECTIMINIMAX}(\text{result}(a, s)) : a \text{ is applicable to } s\}
\]

else return \[
\min \{\text{EXPECTIMINIMAX}(\text{result}(a, s)) : a \text{ is applicable to } s\}
\]

This gives optimal play (i.e., highest expected utility)
With nondeterminism, exact values do matter

At “chance” nodes, we need to compute weighted averages.

Behavior is preserved only by positive linear transformations of Eval.

Hence Eval should be proportional to the expected payoff.
**In practice**

Dice rolls increase \( b \): 21 possible rolls with 2 dice

Given the dice roll, \( \approx 20 \) legal moves on average

(for some dice roles, can be much higher)

\[
\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9
\]

As depth increases, probability of reaching a given node shrinks

\( \Rightarrow \) value of lookahead is diminished

\( \alpha-\beta \) pruning is much less effective

**TDGammon** uses depth-2 search + very good **Eval**

\( \approx \) world-champion level
Summary

We looked at games that have the following characteristics:
  - two players
  - zero sum
  - perfect information
  - deterministic
  - finite

In these games, can do a game-tree search
  - minimax values, alpha-beta pruning

In sufficiently complicated games, perfection is unattainable
  ⇒ must approximate: limited search depth, static evaluation function

In games that are even more complicated, further approximation is needed
  ⇒ Monte Carlo roll-outs

If we add an element of chance (e.g., dice rolls), expectiminimax